Estimation of Cache Related Migration Delays for Multi-Core Processors with Shared Instruction Caches

Damien Hardy, Isabelle Puaut

In Proc. of the 17th International Conference on Real-Time and Network Systems
RTNS'2009, Paris, ECE, 26-27 October, 2009

Estimation of Cache Related Migration Delays for Multi-Core Processors with Shared Instruction Caches

Damien Hardy Isabelle Puaut
Université Européenne de Bretagne / IRISA, Rennes, France

Abstract

Multi-core architectures, which have multiple processors on a single chip, have been adopted by most chip manufacturers. In most such architectures, the different cores have private caches and also shared on-chip caches. For real-time systems to exploit multi-core architectures, it is required to obtain both tight and safe estimations of a number of metrics required to validate the system temporal behaviour in all situations, including the worst-case: tasks worst-case execution times (WCET), preemption delays and migration delays. Estimating such metrics is very challenging because of the possible interferences between cores due to shared hardware resources such as shared caches, memory bus, etc.

In this paper, we propose a new method to estimate worst-case cache reload cost due to a task migration between cores. Safe estimations of the so-called Cache-Related Migration Delay (CRMD) are obtained through static code analysis. Experimental results demonstrate the practicality of our approach by comparing predicted worst-case CRMDs with those obtained by a naive approach. To the best of our knowledge, our method is the first one to provide safe upper bounds of cache-related migration delays in multi-core architectures with shared instruction caches.

1 Introduction

Most chip manufacturers have adopted multi-core technologies to both continue performance improvements and control heat and thermal issues. In most multi-core architectures, the different cores have private caches and also shared on-chip caches.

For real-time systems to exploit multi-core architectures, it is required to obtain both tight and safe estimations of a number of metrics required to validate the system temporal behaviour in all situations, including the worst-case:

- tasks worst-case execution times (WCET), for each task considered in isolation,
- worst-case preemption delays, including the time required to refill the architecture caches after a preemption,
- worst-case migration delays, including the time to reload the missing information into the caches after a migration.

Estimating such metrics is very challenging because of the possible interferences between cores due to shared hardware resources such as shared caches, memory bus, etc.

In this paper, we propose a new method to estimate the worst-case cache reload cost due to the migration of a task between cores. Such a delay is called hereafter CRMD for *Cache Related Migration Delay*. CRMD is due to the cache refill activity occurring after a migration, and is illustrated below in Figure 1. Figure 1 depicts the impact of task migration on the contents of private and shared caches in a multi-core platform. The depicted platform is made of C cores, each having a private L1 instruction cache and a shared L2 instruction cache.

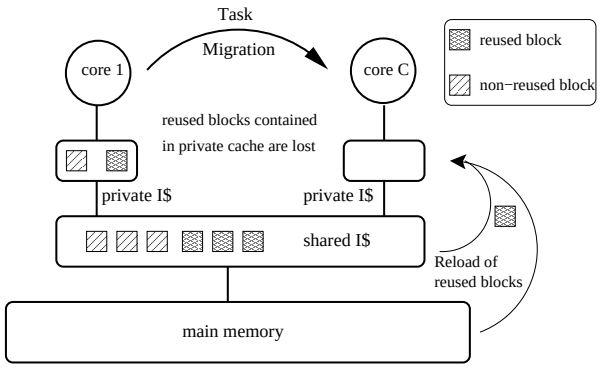


Figure 1. Impact of task migration on cache contents

Consider a task, initially running on *Core 1*, which migrates on *Core C*. At the migration point (see Fig. 1), both the private and the shared instruction caches contain some program blocks. Some program blocks, termed *reused*

blocks, will be used after the task migration, whereas some other blocks, termed *non-reused blocks*, will not be reused after the task migration. After the migration, all reused cache blocks will be reloaded in all levels of the cache hierarchy.

Task migration thus results in additional cache misses compared to a migration-free execution. Such cache misses occur in the private cache, to load reused blocks. They may also occur in the shared cache in case a reused block has been evicted after first loaded, which can occur when using non inclusive cache hierarchies.

Since exact migration points are not statically known, a task migration may result in additional cache accesses in the shared caches compared to a migration-free execution. We chose to account for these accesses when estimating the WCET of a task. Such a WCET, called *migration-aware WCET* assumes that the task may migrate and thus may cause additional accesses to the shared caches, but does not include the cache reload cost itself. Additional cache misses (in the private L1 cache and shared cache levels) are hereafter called *Cache Related Migration Delay (CRMD)*.

We propose in this paper methods to compute safe estimations of the *migration-aware WCET* and the *Cache Related Migration Delay (CRMD)*, using static analysis of the code of the task subject to migration. Experimental results demonstrate that estimated CRMDs are much lower than when using a naive approach assuming that all useful blocks must be reloaded in all cache levels after a migration.

Contributions. The paper contains two tightly-coupled contributions:

- The first contribution is the proposal of a *migration-aware cache analysis method*. The method estimates the worst-case number of cache hits/misses of an isolated task running on a multi-core platform and subject to migrations, regardless of the number of migrations it will suffer at run-time. The proposed migration-aware cache analysis method accounts for every possible migration point on the shared cache, but does not integrate the impact of the cache-related migration cost itself.

- The second contribution is a method to compute a *tight upper bound of the cache-related migration delay (CRMD)* an isolated task will suffer after each migration to reload the reused cache blocks. The provided CRMD is tight because the CRMD does not consider as misses the accesses that are already detected as misses by the migration-aware cache analysis method. This metric, together with the migration-aware WCET estimate, provides a safe bound of cache-related migration costs in a multi-core system. It can be used in any real-time multi-processor schedulability test for global and semi-partitioned scheduling [3, 2, 14] to the extent that the worst-case number migrations is known.

To the best of our knowledge, our method is the first one to provide safe upper bounds of cache refill costs in case of migrations for multi-core architectures with shared instruction caches. This approach focuses on the computation of the CRMD of a task in isolation and has to be used in combination with a cache-related preemption delays estimation method [20, 25].

Related work. Many static WCET estimation methods have been designed in the last two decades (see [28] for a survey). Static WCET estimation methods need a low-level analysis phase to determine the worst-case timing behavior of the micro-architectural components (pipelines and out-of-order execution, branch predictors, caches, etc.). Regarding cache memories on mono-core architectures, two main classes of approaches have been proposed: *static cache simulation* [18, 19], based on dataflow analysis, and the methods described in [9, 26, 10], based on abstract interpretation. Both classes of methods provide for every memory reference a classification of the outcome of the reference in the worst-case execution scenario (e.g. *always-hit*, *always-miss*, *first-miss*, etc.). These methods, originally designed for code only, and for direct-mapped or set-associative caches with a Least Recently Used (LRU) replacement policy, have been later extended to other replacement policies [13], data and unified caches [27], and caches hierarchies [12]. Cache-aware WCET estimation methods have recently been extended to multi-core platforms [29, 11]; the cited methods take into account the interferences caused by shared caches. The proposed method for evaluating migration-aware WCETs is based on [12], itself based on abstract interpretation for static cache analysis [9, 26, 10].

The presence of caches not only impacts the execution time of tasks considered in isolation but also results in an indirect cost required to refill the caches after a preemption. Static analysis techniques, close to those designed for cache-aware WCET estimation, aim at producing safe upper bound of CRPDs (cache-related preemption delays) [20, 25]. Such techniques statically analyze the code of the preempted and preempting tasks to determine which blocks from the preempted task may be reused after the preemption and will have to be reloaded. The method we propose to evaluate CRMDs uses similar analyses and data structures as the ones used to estimate CRPDs.

Extensive empirical evaluations of the impact of real-world overheads (including cache-related preemption and migration overheads) on multiprocessor scheduling algorithms have been presented in [5, 4]. In contrast to our work, these studies focus on giving average-case and worst-

measured overheads and do not aim at providing safe upper bounds of cache-related overheads.

Cache-aware multi-core scheduling have been presented in [6, 7] for soft real-time applications; the idea of this direction of work is to improve task scheduling in multi-core platforms based on the cache behaviour of real-time tasks. In this paper, we focus on the estimation of cache-related overheads, and consider their exploitation by multiprocessor scheduling algorithms as outside the scope of the paper.

Finally, [24] which is the work closest to ours, assumes a multi-core architecture with a private cache hierarchy. They introduce new hardware support to move the cache contents from one private cache to another to reduce the migration cost. Our approach do not require any hardware modification and the cache hierarchy can be shared between cores except the first cache level.

Paper outline. The rest of the paper is organized as follows. Section 2 presents the assumptions our analysis is based on, regarding the target architecture and task scheduling. Section 3 presents the migration-aware cache analysis method. Section 4 focuses on the estimation of cache-related migration delays. Experimental results are given and discussed in Section 5. Finally, Section 6 gives some conclusions and direction for future work.

2 Assumptions

A multi-core architecture is assumed. Each core has a private first-level (L1) instruction cache, followed by shared instruction cache levels. Each shared cache is shared between all the cores of the architecture. The caches are set-associative and each level of the cache hierarchy is non-inclusive:

- A piece of information is searched for in the cache of level ℓ if and only if a cache miss occurred when searching it in the cache of level $\ell-1$. Cache of level 1 is always accessed.
- Every time a cache miss occurs at cache level ℓ, the entire cache block containing the missing piece of information is always loaded into the cache of level ℓ.
- There are no actions on the cache contents (i.e. invalidations, lookups/modifications) other than the ones mentioned above.

Our study concentrates on instruction caches; it is assumed that the shared caches do not contain data. This study can be seen as a first step towards a general solution for shared caches. It can also push to the use of separate shared instruction and data caches instead of unified ones[1].

[1]Unified caches could be partitioned at boot time for instance in a A-way instruction cache and a B-way data cache.

Our method assumes a LRU (Least Recently Used) cache replacement policy. Furthermore, an architecture without timing anomalies as defined in [16] is assumed. The access time variability to main memory and shared caches, due to bus contention, is supposed to be bounded and known, by using for instance *Time Division Multiple Access (TDMA)* like in [23] or other predictable bus arbitration policies [21]. Figure 2 illustrates two different supported architectures.

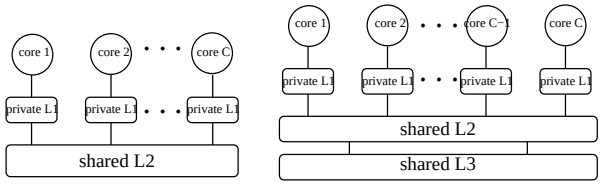


Figure 2. Two examples of supported architectures

The focus in this paper is to estimate the worst-case *cache related migration delay (CRMD)* suffered from a hard real-time task after a migration from one core to another in a multi-core platform. The migrated task is considered in isolation from the tasks running at the same time on the multi-core platform. The computation of interferences due to intra-core or inter-core of other tasks is considered out of the scope of this paper; for related studies tackling these issues, the reader is referred to [20, 25] regarding intra-core interferences or [29, 11] regarding inter-core interferences.

3 Migration-aware multi-level cache analysis

As a first step to present the *migration-aware cache analysis* method, paragraph 3.1 focuses on the analysis of the worst-case behaviour of the memory hierarchy when completely ignoring task migrations, what we call *migration-ignorant cache analysis*. The impact of task migration on shared caches is considered in paragraph 3.2.

3.1 Migration-ignorant cache analysis

The cache analysis, originally presented in [12] and briefly described is applied successively on each level of the cache hierarchy, from the first cache level to the main memory. The analysis is contextual in the sense that it is applied for every call context of functions (functions are virtually inlined). The references considered by the analysis of cache level ℓ depend on the outcome of the analysis of cache level $\ell-1$ to consider the filtering of memory accesses between cache levels, as depicted in Figure 3 and detailed below.

The outcome of the static cache analysis for every cache level ℓ is a *Cache Hit/Miss Classification (CHMC)* for each

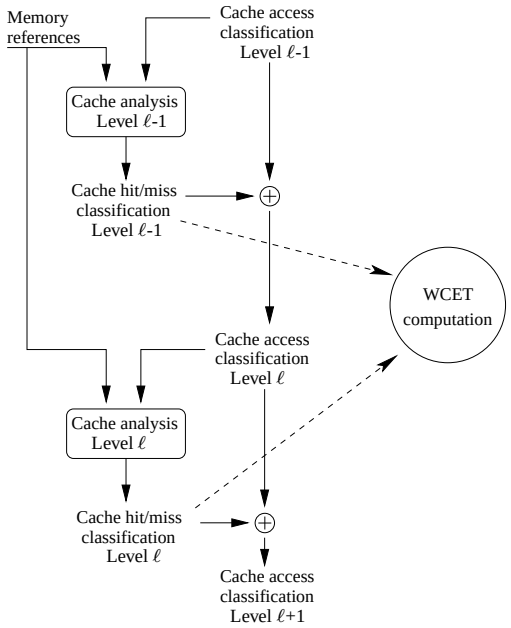


Figure 3. Multi-level cache analysis without task migration

reference, determining the worst-case behavior of the reference with respect to cache level ℓ:

- *always-miss* (AM): the reference will always result in a cache miss,
- *always-hit* (AH): the reference will always result in a cache hit,
- *first-miss* (FM): the reference could neither be classified as hit nor as miss the first time it occurs but will result in cache hit afterwards,
- *not-classified* (NC): in all other cases.

Moreover, at every level ℓ, a *Cache Access Classification (CAC)* specifies if an access may occur or not at level ℓ, and thus should be considered by the static cache analysis of that level. There is a CAC, noted $CAC_{r,\ell,c}$ for every reference r, cache level ℓ, and call context c[2]. The CAC defines three categories for each reference, cache level, and call context:

- A (Always): the access always occurs at cache level ℓ.
- N (Never): the access never occurs at cache level ℓ.
- U (Uncertain) when the access cannot be classified in the two above categories.

The cache analysis at every cache level is based on a state-of-the-art single-level cache analysis [26], based on abstract interpretation. The method is based on three separate fixpoint analyses applied on the program control flow graph, for every call context:

- a *Must* analysis determines if a memory block is always present in the cache at a given point: if so, the block is classified *always-hit (AH)*;
- a *Persistence* analysis determines if a memory block will not be evicted after it has been first loaded; the classification of such blocks is *first-miss (FM)*.
- a *May* analysis determines if a memory block may be in the cache at a given point: if not, the block is classified *always-miss (AM)*. Otherwise, if neither detected as always present by the *Must* analysis nor as persistent by the *Persistence* analysis, the block is classified *not classified (NC)*;

Abstract cache states (ACS) are computed for every basic block according to the semantics of the analysis (Must/May/Persistence) and the cache replacement policy by using functions ($Update$ and $Join$) in the abstract domain. $Update$ models the impact on the ACS of every reference inside a basic block; $Join$ merges two ACS at convergence points in the control flow graph (e.g. at the end of conditional constructs).

Figure 4 gives an example of an ACS of a 2-way set-associative cache with LRU replacement policy on a *Must* analysis (only one cache set is depicted). An *age* is associated to every cache block of a set. The smaller the block age the more recent the access to the block. For the *Must* analysis, each memory block is represented only once in the ACS, with its maximum age. It means that its actual age at run-time will always be lower than or equal to its age in the ACS.

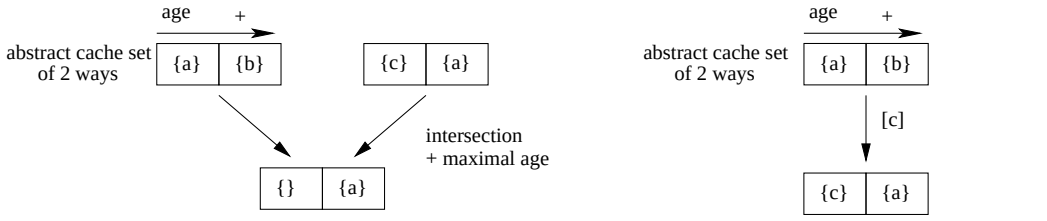
At every cache level ℓ, the three analyses (*Must, May, Persistence*) consider all references r guaranteed to occur at level ℓ ($CAC_{r,\ell} = A$). References with $CAC_{r,\ell} = N$ are not analysed. Regarding uncertain references ($CAC_{r,\ell} = U$), for the sake of safety, the ACS is obtained by exploring the two possibilities ($CAC_{r,\ell} = A$ and $CAC_{r,\ell} = N$) and merging the results using the $Join$ function (see Figure 5). For all references r, $CAC_{r,1} = A$, meaning that the L1 cache is always accessed.

Since task migrations are not considered in this paragraph, the CAC of a reference r for a cache level ℓ only depends on CHMC of r at level $\ell-1$ and the CAC of r at level $\ell-1$ to model the filtering of accesses in the cache hierarchy (see Figure 3). Table 1 shows all the possible cases of computation of $CAC_{r,\ell}$ from $CHMC_{r,\ell-1}$ and $CAC_{r,\ell-1}$.

$CAC_{r,\ell-1}$ \ $CHMC_{r,\ell-1}$	AM	AH	FM	NC
A	A	N	U	U
N	N	N	N	N
U	U	N	U	U

Table 1. Cache access classification for level ℓ ($CAC_{r,\ell}$)

[2]The call context c will be omitted from the formulas when the concept of call context is not relevant.



a. Join function of Must analysis

b. Update function of Must analysis

Figure 4. $Join$ **and** $Update$ **functions for the Must analysis with LRU replacement**

The CHMC of reference r is used to compute the cache contribution to the WCET of that reference (i.e. the sum of the cache level latencies where the access to r may occur plus the memory latency if the access may occur in the memory), which can be included in well-known WCET computation methods [17, 22].

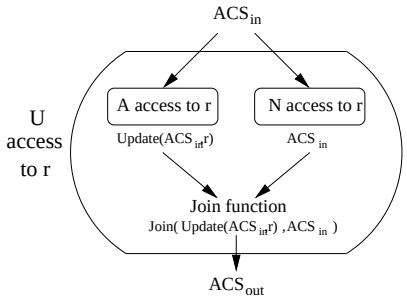


Figure 5. Function for U access

3.2 Migration-aware cache analysis

As previously depicted in Figure 1, migrating a task results in additional accesses to the shared caches after the migration. Since the exact migration points are not known off-line, some accesses to the shared cache levels that would not occur in a migration-free execution may occur after the migration. Thus, our migration-aware cache analysis account for every possible migration point without integrating the cache-related migration cost itself.

As it was previously demonstrated in [12], considering these additional accesses to the shared caches as always occurring might not be safe, because this can lead to an underestimation of the reuse distance of blocks in the shared caches. As a consequence, the migration-aware cache analysis considers all accesses to the first shared cache level (usually L2 cache) as *Uncertain* ($CAC_{r,\ell} = U$ with ℓ the first shared cache level). This ensures a safe cache analysis of the shared cache levels in the presence of unknown migration points. Apart from the introduction of U accesses in the first shared cache level, the cache analysis and computation of migration-aware WCETs, noted hereafter $WCET_{MA}$ are achieved as described in § 3.1.

Note that $WCET_{MA}$ is more pessimistic than its migration-ignorant counterpart. This is because $WCET_{MA}$ accounts for the impact of migrations on the shared cache(s), which are not accounted for when estimating the migration-ignorant WCET. The additional pessimism due to the consideration of possible task migrations is evaluated in Section 5.

4 Computing Cache-Related Migration Delay (CRMD)

This section focuses on the computation of the Cache-Related Migration Delay (CRMD) suffered by a task τ every time it migrates from one core to another. When τ migrates n times, its WCET is then:

$$WCET_{MA} + n * (CRMD + \delta)$$

with δ the migration cost excluding cache reloads. The maximum number of migrations suffered by a task at runtime depends on the scheduling policy[3].

Due to the use of the migration-aware cache analysis, the CRMD only depends on the additional accesses to the memory hierarchy after the migration. As explained before, and previously illustrated in Figure 1, extra accessed concern blocks reused after the migration of τ, and may introduce additional misses in the L1 cache as well as in the shared cache levels.

Useful cache blocks. To bound the number of reused blocks of the L1 cache at each program point, we use the notion of *useful cache blocks* previously defined in [15] for the computation of Cache-Related Preemption Delay (CRPD). A useful cache block is defined as follows: *a useful cache block at an execution point is defined as a memory block that may be re-referenced before being replaced*. In other

[3]This estimation is independent from any scheduling policies. It can be reduced by considering the n highest values of the CRMD instead of n times the maximal value with some extra restrictions on the migration points.

words, the set of useful cache blocks at a given program point p is a safe over-approximation of the set of reused blocks at program point p.

The technique used to determine the useful cache blocks is based on the traditional *reaching definitions* and *live variables* data flow analyses [1]:

- Similarly to the reaching definitions analysis, the *reaching memory blocks* (RMB) analysis determines all the memory blocks that may be in the cache at a program point p when p is reached via any incoming program path.
- As in the live variables analysis, the *live memory blocks* (LMB) analysis determines all the memory blocks that may be referenced before their eviction via any outgoing path from p.

The useful cache blocks at program point p (noted $useful(p)$) are the memory blocks that are present in the result of both the RMB analysis (noted $RMB(p)$) and the LMB analysis (noted $LMB(p)$).

$$useful(p) = RMB(p) \cap LMB(p)$$

Suppose that task τ migrates at program point p. Instead of considering a miss in *all* cache levels for each useful cache block at point p, our computation produces tighter results by integrating in the CRMD only misses which are not already integrated in the migration-aware WCET estimate.

Notations. Before detailing the computation of the CRMD, let us introduce some formulae obtained from the results of the migration-aware cache analysis. First, we introduce the notion of *always-persistent* block to determine if a cache block cb is ensured to hit after a migration in a given shared cache level ℓ (i.e. its cache hit/miss classification is *always-hit* or *first-miss* in all execution contexts):

$$always\_persistent_\ell(cb) =$$

$$\begin{cases} true & \text{if } \forall_{ctx}, \forall_{instr} \in cb, \\ & \quad CHMC_{\ell,ctx}(instr) = AH \\ & \quad \vee CHMC_{\ell,ctx}(instr) = FM \\ false & \text{otherwise} \end{cases}$$

We also define the notion of *always-filtered* block by a previous shared cache level(s) of ℓ if the cache block cb is always-persistent in at least one previous shared cache level:

$$always\_filtered_\ell(cb) =$$

$$\begin{cases} false \text{ if } \ell = 2 \\ \bigvee_{p\ell=2}^{\ell-1} always\_persistent_{p\ell}(cb) \text{ otherwise} \end{cases}$$

Similarly, we introduce $at\_least\_once\_persistent_\ell(cb)$ to detect the case where a cache block cb produces a hit in shared cache level ℓ in at least one execution context:

$$at\_least\_once\_persistent_\ell(cb) =$$

$$\begin{cases} true & \text{if } \exists_{ctx}, \exists_{instr} \in cb, \\ & \quad CHMC_{\ell,ctx}(instr) = AH \\ & \quad \vee CHMC_{\ell,ctx}(instr) = FM \\ false & \text{otherwise} \end{cases}$$

and $at\_least\_once\_filtered_\ell(cb)$ by a previous shared level(s) of ℓ if the cache block cb is at-least-once-persistent in at least one previous shared level:

$$at\_least\_once\_filtered_\ell(cb) =$$

$$\begin{cases} false \text{ if } \ell = 2 \\ \bigvee_{p\ell=2}^{\ell-1} at\_least\_once\_persistent_{p\ell}(cb) \\ \quad \text{otherwise} \end{cases}$$

Finally, we define *private-filtered* to determine if a cache block is completely filtered by the private L1 cache in at least one execution context during the computation of $WCET_{MA}$:

$$private\_filtered(cb) = \\ \exists ctx, \forall instr \in cb, CHMC_{L1,ctx}(instr) = AH$$

Computing CRMD. A miss that could occur for the first reference in the case of a *first-miss* is already counted by the cache-aware migration analysis and there is no need to count it twice except in the case the access is private-filtered.

The L1 cache is always accessed, thus the latency of the L1 cache is already included in $WCET_{MA}$ and do not need to be counted in the CRMD. For a given shared cache level ℓ, an access to a useful cache block ucb after a migration has to be counted if the access is private-filtered because in this case, the access could be not have been counted during $WCET_{MA}$ computation. Moreover, if the access is not private-filtered but this access is not filtered by a previous shared cache level (i.e. $\neg always\_filtered_\ell(ucb)$) and is at-least-once-persistent, the access has to be counted. Remark that if the access is ensured to never produce a hit (i.e. $\neg at\_least\_once\_persistent_\ell(ucb)$), the latency of this access in shared cache level ℓ is already in $WCET_{MA}$. More formally, we define the cost added to the CRMD of a shared cache level ℓ at a given program point p as follows:

$$cost\_share\_level_{\ell}^{p} =| \{ucb \in useful(p), (\neg always\_filtered_{\ell}(ucb) \wedge at\_least\_once\_persistent_{\ell}(ucb)) \vee private\_filtered(ucb)\} | * latency_{\ell}$$

The accesses to the main memory which have to be included in the CRMD are similar. If the access is private-filtered, this access could be not counted during $WCET_{MA}$ computation. Moreover, if the access is not private-filtered but this access is not filtered by any previous shared cache levels (i.e. $\neg always\_filtered_{h\ell+1}(ucb)$ where $h\ell$ represents the level of the highest cache level and $h\ell+1$ represent the level of the main memory) and is at-least-once-filtered by a shared cache level, the main memory latency of the access have to be counted. More formally, we define the cost added to the CRMD of the main memory at a given program point p as follows:

$$cost\_memory^{p} =| \{ucb \in useful(p), (\neg always\_filtered_{h\ell+1}(ucb) \wedge at\_least\_once\_filtered_{h\ell+1}(ucb)) \vee private\_filtered(ucb)\} | * latency_{memory}$$

Thus the CRMD at program point p, noted $CRMD^{p}$ is the sum of the cost of each shared cache level plus the memory cost.

$$CRMD^{p} = cost\_memory^{p} + \sum_{\ell=2}^{h\ell} cost\_share\_level_{\ell}^{p}$$

Finally, the CRMD of one single migration is equal to the biggest value of $CRMD^{p}$ computed for all the program points:

$$CRMD = max(CRMD^{p}, \forall p \in program)$$

5 Experimental results

5.1 Experimental setup

Cache analysis and WCET estimation. The experiments were conducted on MIPS R2000/R3000 binary code compiled with gcc 4.1 with no optimization and with the default linker memory layout. The WCETs of tasks are computed by the Heptane timing analyzer [8], more precisely its Implicit Path Enumeration Technique (IPET). The analysis is context sensitive (functions are analysed in each different calling context). To separate the effect of the caches from those of the other parts of the processor micro-architecture, WCET estimation only takes into account the contribution of instruction caches to the WCET. The effects of other architectural features are not considered. In particular, timing anomalies caused by interactions between caches and pipelines, as defined in [16] are disregarded. The cache classification *not-classified* is thus assumed to have the same worst-case behavior as *always-miss* during the WCET computation in our experiments. For space consideration, WCET computation is not detailed here, interested readers are referred to [12].

The migration points considered in the experiments are the ends of basic blocks of the analyzed task.

Name	**Description**	**Code size** (bytes)
crc	Cyclic redundancy check computation	1432
fft	Fast Fourier Transform	3536
jfdctint	Integer implementation of the forward DCT (Discrete Cosine Transform)	3040
matmult	Multiplication of two 50x50 integer matrices	1200
minver	Inversion of floating point 3x3 matrix	4408
adpcm	Adaptive pulse code modulation algorithm	7740
statemate	Automatically generated code by STARC (STAtechart Real-time-Code generator)	8900

Table 2. Benchmark characteristics

Benchmarks. The experiments were conducted on seven benchmarks (see Table 2 for the applications characteristics). All benchmarks are maintained by Mälardalen WCET research group[4].

Cache hierarchy. The results are obtained on a 2-level cache hierarchy composed of a private 4-way L1 cache of 1KB with a cache block size of 32B and a shared 8-way L2 cache of 2KB (or 4KB for the two biggest benchmarks *adpcm* and *statemate*) configured with a cache block size of 32B or 64B. Cache sizes are small compared to usual cache sizes in multi-core architectures. However, there are no large-enough public real-time benchmarks available to experiment our proposal. As a consequence, we have selected quite small commonly used real-time benchmarks and adjusted cache sizes such that the benchmarks do not fit entirely in the caches. All caches are implementing a LRU replacement policy. Latencies of 1 cycle (respectively 10 and 100 cycles) are assumed for the L1 cache (respectively the L2 cache and the main memory).

[4] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

5.2 Results

First, the overestimation resulting from accounting for possible migration points when estimating the WCET of tasks is estimated. Then, the CRMD estimated using our method is compared to a baseline CRMD estimation method considering that all useful blocks are reloaded in all cache levels after a task migration. Finally, the execution time of CRMD estimation is evaluated.

Impact of migrations on task WCET for a non-migrating task. In this paragraph, we focus on the comparison of the estimated migration-ignorant WCET (noted $WCET_{MI}$) and the migration-aware WCET (noted $WCET_{MA}$) when the task does not migrate. The results are mainly given in Table 3, which shows the WCET overestimation in cycles resulting from considering every possible migration point. More details regarding the results of cache analysis are given in Table 4.

Benchmarks	WCET$_{MI}$ (cycles)	WCET$_{MA}$ (cycles)	delta (cycles)	ratio
crc (2KB-32B)	152753	152753	0	0%
crc (2KB-64B)	151953	152753	800	0.53%
fft (2KB-32B)	188655	188655	0	0%
fft (2KB-64B)	187555	188655	1100	0.59%
jfdctint (2KB-32B)	25389	25389	0	0%
jfdctint (2KB-64B)	20689	25389	4700	22.72%
matmult (2KB-32B)	16704	16704	0	0%
matmult (2KB-64B)	16504	16704	200	1.21%
minver (2KB-32B)	20646	20646	0	0%
minver (2KB-64B)	16446	20646	4200	25.54%
adpcm (4KB-32B)	310391	316391	6000	1.93%
adpcm (4KB-64B)	322125	383439	61314	19.03%
statemate (4KB-32B)	141303	142603	1300	0.92%
statemate (4KB-64B)	115903	152325	36422	31.42%

Table 3. Migration-ignorant WCET vs migration-aware WCET

We observe from Table 4 three different situations, which allows to explain the results given in Table 3.

- The first situation is when the migration-ignorant cache analysis does not detect any hit in the L2 cache, or detects very few hits in the L2 cache (in Table 4 number of L1 misses $\approx$ number of L2 misses). This situation occurs when the migration-ignorant cache analysis does not detect spatial and temporal locality in the L2 cache. In this situation, the migration-aware WCET is very close to the migration-ignorant WCET.
- The second situation occurs when the migration-ignorant cache analysis detects temporal locality but no spatial locality in the L2 cache (in Table 4 number of L1 misses $\gg$ number of L2 misses, with L2 cache lines of 32B). In this situation, the migration-aware

Benchmarks	Metrics	Migration-ignorant	Migration-aware
crc (2KB-32B)	nb of L1 accesses	141643	141643
	nb of L1 misses	101	101
	nb of L2 misses	101	101
crc (2KB-64B)	nb of L1 accesses	141643	141643
	nb of L1 misses	101	101
	nb of L2 misses	93	101
fft (2KB-32B)	nb of L1 accesses	80305	80305
	nb of L1 misses	7575	7575
	nb of L2 misses	326	326
fft (2KB-64B)	nb of L1 accesses	80305	80305
	nb of L1 misses	7575	7575
	nb of L2 misses	315	326
jfdctint (2KB-32B)	nb of L1 accesses	8039	8039
	nb of L1 misses	725	725
	nb of L2 misses	101	101
jfdctint (2KB-64B)	nb of L1 accesses	8039	8039
	nb of L1 misses	725	725
	nb of L2 misses	54	101
matmult (2KB-32B)	nb of L1 accesses	11204	11204
	nb of L1 misses	50	50
	nb of L2 misses	50	50
matmult (2KB-64B)	nb of L1 accesses	11204	11204
	nb of L1 misses	50	50
	nb of L2 misses	48	50
minver (2KB-32B)	nb of L1 accesses	4146	4146
	nb of L1 misses	150	150
	nb of L2 misses	150	150
minver (2KB-64B)	nb of L1 accesses	4146	4146
	nb of L1 misses	150	150
	nb of L2 misses	108	150
adpcm (4KB-32B)	nb of L1 accesses	186301	186301
	nb of L1 misses	3759	3759
	nb of L2 misses	865	925
adpcm (4KB-64B)	nb of L1 accesses	186435	186569
	nb of L1 misses	3779	3797
	nb of L2 misses	976	1589
statemate (4KB-32B)	nb of L1 accesses	10933	10933
	nb of L1 misses	1797	1797
	nb of L2 misses	1124	1137
statemate (4KB-64B)	nb of L1 accesses	10673	10945
	nb of L1 misses	1763	1798
	nb of L2 misses	876	1239

Table 4. Migration-ignorant vs migration-aware cache analysis (estimated number of accesses)

WCET is still close to the migration-ignorant WCET. The good result comes from the presence of the persistence analysis, which detects blocks as persistent even though accesses to the L2 cache are considered as *Uncertain* (U).

- Finally, the third and last situation occurs when the migration-ignorant cache analysis detects both temporal and spatial locality in the L2 cache (in Table 4 number of L1 misses $\gg$ number of L2 misses, with L2 cache lines of 64B). In this situation, the migration-aware WCET might be significantly larger than is migration-ignorant counterpart. This is because the introduction of U accesses in the migration-aware cache analysis prevents the cache analysis from detecting spatial locality in the L2 cache.

It can be remarked that there are for some benchmarks (*adpcm* and *statemate*) a variation of worst-case execution path between the migration-aware and migration-ignorant cases (different numbers of accesses along the worst-case execution path for the L1 cache).

Benchmarks	# useful cache block	CRMD baseline in cycles	CRMD in cycles
crc (2KB-32B)	31	3410	510
crc (2KB-64B)	31	3410	400
fft (2KB-32B)	32	3520	1050
fft (2KB-64B)	32	3520	610
jfdctint (2KB-32B)	20	2200	460
jfdctint (2KB-64B)	20	2200	360
matmult (2KB-32B)	17	1870	190
matmult (2KB-64B)	17	1870	140
minver (2KB-32B)	14	1540	280
minver (2KB-64B)	14	1540	240
adpcm (4KB-32B)	24	2640	970
adpcm (4KB-64B)	24	2640	690
statemate (4KB-32B)	5	550	20
statemate (4KB-64B)	5	550	110

Table 5. Estimated Cache-Related Migration Delay (CRMD)

Evaluation of CRMD. Table 5 compares for every benchmark and cache configuration the CRMD obtained by our proposed method (column 4) to a simple baseline, albeit safe method considering that all useful blocks have to be reloaded in all cache levels after a task migration (column 3). Column 2 gives the number of useful cache blocks per benchmark and cache configuration.

The numbers given in the table show that the estimated CRMD, obtained by the proposed approach, is much lower than when using the simple baseline approach. Comparing estimated CRMD with measured ones is left for future work.

5.2.1 Analysis time.

The longest measured time to estimate the migration-aware WCET plus to estimate the CRMD was 5 minutes for the biggest benchmarks. This shows empirically that the complexity of CRMD estimation is similar to the one of cache analyses used when estimating WCETs.

6 Conclusions and future work

We have proposed in this paper a new method, based on static analysis, to estimate the worst-case cache reload cost due to the migration of a task between cores (CRMD, for Cache Related Migration Delay). To the best of our knowledge, our method is the first one to provide *safe* upper bounds of cache-related migration delays in multi-core architectures with shared caches. Experimental results have shown that the estimated CRMDs are much less pessimistic than the simple baseline safe approach except when the cache block sizes in the different cache levels are not the same.

As future work, we plan to compare the estimated CRMDs with measured ones in order to evaluate the tightness of our approach. Other research directions will be to extend the approach to data or unified caches. Finally, selecting task scheduling based on CRMD information would be of interest.

Acknowledgments. The authors are grateful to Benjamin Lesage and to the anonymous reviewers for feedback on earlier versions of the paper.

References

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, GB, 1988.

[2] S. K. Baruah and T. P. Baker. Schedulability analysis of global EDF. *Real-Time Systems*, 38(3):223–235, 2008.

[3] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[4] A. Block, B. Brandenburg, J. Anderson, and S. Quint. An adaptive framework for multiprocessor real-time system. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 23–33, July 2008.

[5] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 157–169, Washington, DC, USA, 2008. IEEE Computer Society.

[6] J. Calandrino and J. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 299–308, July 2008.

[7] J. Calandrino and J. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, July 2009.

[8] A. Colin and I. Puaut. A modular and retargetable framework for tree-based WCET analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 37–44, Delft, The Netherlands, June 2001.

[9] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[10] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for real-life processor. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485, Tahoe City, CA, USA, Oct. 2001.

[11] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th Real-Time Systems Symposium*, Washington D.C., USA, Dec. 2009. To appear.

[12] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 29th Real-Time Systems Symposium*, pages 456–466, Barcelona, Spain, Dec. 2008.

[13] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE, vol.9, n7*, 2003.

[14] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *In Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS2009)*, San Francisco, CA, USA, April 2009.

[15] C.-G. Lee, H. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computer*, 47(6):700–713, June 1998.

[16] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium*, pages 12–21, 1999.

[17] S. Malik and Y. T. S. Li. Performance analysis of embedded software using implicit path enumeration. *Design Automation Conference*, 0:456–461, 1995.

[18] F. Mueller. *Static cache simulation and its applications*. PhD thesis, Florida State University, 1994.

[19] F. Mueller. Timing analysis for instruction caches. *Real Time Systems*, 18(2-3):217–247, 2000.

[20] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 201–206, 2003.

[21] M. Paolieri, E. Qui nones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 57–68, New York, NY, USA, 2009. ACM.

[22] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real Time Systems*, 1(2):159–176, 1989.

[23] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 49–60, Washington, DC, USA, 2007. IEEE Computer Society.

[24] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan. Push-assisted migration of real-time tasks in multi-core processors. *SIGPLAN Not.*, 44(7):80–89, 2009.

[25] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 278–286, 2004.

[26] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real Time Systems*, 18(2-3):157–179, 2000.

[27] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real Time Systems*, 17(2-3):209–233, 1999.

[28] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.

[29] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared l2 instruction caches. In *RTAS '08: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89, 2008.